CONSTANT EXPRESSIONS IN C++

Professor Ken Birman
CS4414 Lecture 9

IDEA MAP FOR TODAY

Many things can be “precomputed”. In C++ these include the loop bounds we saw in lecture 8, the types used in “auto” declarations, and code-inlining and refactoring

The C++ compiler is especially famous for this

Why is this technique so valuable?

How do other languages handle the same issue?

Keywords: const, constexpr, consteval

CONNECTION TO CONCEPTUAL ABSTRACTION

As we saw in prior lectures, abstract thinking isn't limited just to designing ADTs and modules implementing data structures

Dijkstra taught us to think about abstractions that might span entire layers of the operating system (like file systems). We have seen how by understanding those abstractions, we gain valuable forms of *control*. Constexpr is an abstract tool, too!

HOW DO PROGRAMS IN C OR C++ BECOME EXECUTABLES?

Languages like Python and Java are highly portable. They compile to byte code… Java does "just in time" compilation to machine code. A JIT or interpreter must be rapid.

In contrast, C++ is compiled using an optimization-driven model: *on this architecture, what is the best way to turn your code into machine instructions?* It is willing to spend a lot of time during compilation to reduce delay at runtime.

CONSIDER THE HUMBLE PROCEDURE CALL…

In fact, let’s look at an example:

fibonacci(n) computes the n’th fibonacci integer

```
int fibonacci(int n)
{
    if(n <= 1)
        return n;
    return fibonacci(n-1)+fibonacci(n-2);
}
```

$$0 \quad 1 \; 1 \; 2 \; 3 \; 5 \; \underbrace{8 \; 13}_{21 \,=\, 8 \,+\, 13} \; 21 \; \ldots.$$

… FIBONACCI IS THE MOST FAMOUS EXAMPLE OF RECURSION

When first introduced to recursion, many students are confused because

1. The method is invoking itself,
2. The variable n is being used multiple times in different ways,
3. We even call fibonacci twice in the same block!

Over time, you learn to think in terms of "scope" and to view each instance as a separate scope of execution.

WHERE IS FIBONACCI PROCESSED?

In the case of Java or Python, we would know that Fibonacci is performed at runtime, and we learn all about the costs (will review these costs in a moment).

But the bottom line is: The function is translated to a highly efficient data structure (Python) or intermediate code that maps to instructions (Java), then this logic is interpreted or executed.

WHERE IS FIBONACCI PROCESSED?

In C++ there are several possible answers.

The compiler generates any required code but with a more complete analysis: Python and Java types cannot be fully known until runtime, whereas C++ types are known at compile time.

But there are also cases where *less* code or *no* code is needed!

… DOES N NEED A MEMORY LOCATION?

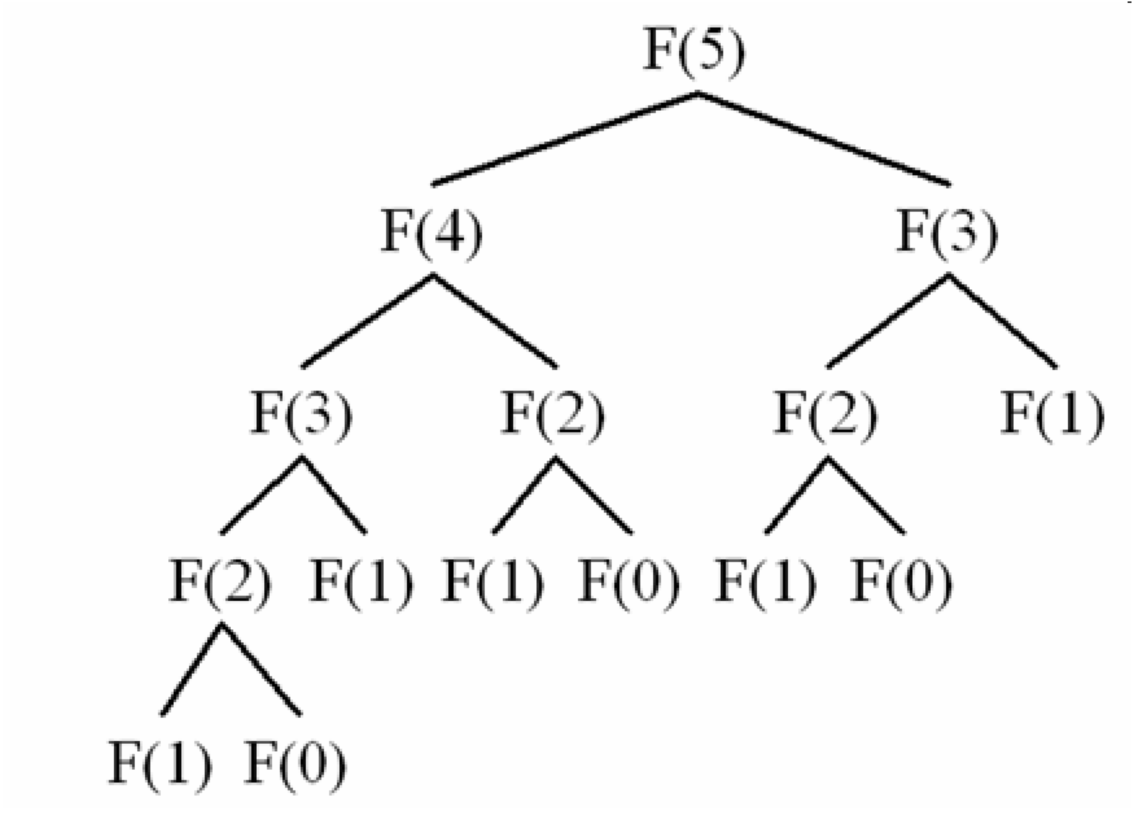
Where does the memory for argument n reside? In Java or Python, n resides on the stack. Each time fibonacci is called:

- Push any registers to the stack, including the return PC
- Push arguments (in our case, the current value of n)
- Jump to fibonacci, which allocates space on the stack for local variables (in our case there aren't any), and executes
- When finished, fibonacci pops the PC and returns to the caller
- The caller code pops data it pushed (and perhaps also the result)

FIBONACCI(5)

```
int fibonacci(int n)
{
    if(n <= 1)
        return n;
    return fibonacci(n-2)+fibonacci(n-1);
}
```

```
fibonacci(5) = fibonacci(3)+Fibonacci(4)
fibonacci(4) = fibonacci(2)+Fibonacci(3)
fibonacci(3) = fibonacci(1)+Fibonacci(2)
fibonacci(2) = fibonacci(0)+Fibonacci(1)
fibonacci(1) = 1
Fibonacci(0) = 1
```



Due to repeatitive pattern, requires 15 calls to Fibonacci!

COMMON OPTIMIZATION

In CS2110 we teach about caching (memoization). But a compiler would not automate this solution: it “changes the code”

```
int fibonacci(n)
{
   if(n <= 1)
       return n;
    if(!known_results.contains(n)) {
       known_results[n] = fibonacci(n-1)+fibonacci(n-2);
    }
    return known_results[n];
}
```

WITHOUT MEMOIZATION, WHERE IS TIME BEING SPENT?

How many instructions really relate to computing fibonacci?

2

We have an if statement: a comparison (call it compare "a and b") then branch "if a >= b".

1

Two recursive calls, one addition, then return.

2 * ? + 1 + 1

THE COST OF THE RECURSIVE CALLS?

They each

- Push registers. Probably 1 is in use. 1
- Push arguments. In our case, value of n. 1
- Push the return PC, jump to fibonacci 2
- After the call, we need to pop the arguments and also pop the saved registers. 2

… NOW WE CAN FILL IN THE "?" WITH 6

How many instructions really relate to computing fibonacci? 2

We have an if statement: a comparison (call it compare "a and b") then branch "if a >= b". 1

Two recursive calls, one addition, then return. 2 * 6 + 2 + 1

HOW MANY INSTRUCTIONS TO PUSH AND POP ARGUMENTS?

About 15 instructions per call to fibonacci. Of these, 1 is the actual addition operation, and the others are “housekeeping”

For example: fibonacci(5)=0…1…1…2…3…5

Our code needs to do the required 5 additions. However, to compute it we will do 15 recursive calls at a cost of about 15 instructions each: 255 instructions… 51x slower than ideal!

SOME QUESTIONS WE CAN ASK

When C++ creates space for us to hold n on the stack, why is it doing this?

We should have a copy of n if we will make changes, but then would want them discarded, or perhaps if the caller might be running a concurrent thread that could make changes to n "under our feet" (if the caller is spawning concurrent work).

But Fibonacci does not change n!

C++ "CONST" ANNOTATION

Expresses the promise that something will not be changed. (At least, won't be changed by this piece of code).

The compiler can then use that knowledge to produce better code, in situations where an opportunity arises.

Can only be used if you genuinely won't change the value!

FIBONACCI WITH CONST

Our code doesn't change n, so we could try:

```
int fibonacci(const int& n)
{
    if(n <= 1)
        return n;
    return fibonacci(n-1)+fibonacci(n-2);
}
```

… but n-1 and n-2 aren't guaranteed to be in memory, so C++ will complain that it can't make a reference to them!

C++ CONST ANNOTATION

The easiest case:

```
const int  MAXD = 1000;       // Length of myvec
char  myvec[MAXD];             //  digits is an array 8-bit ints
```

Here, we are declaring a "compile time constant". C++ knows that MAXD is constant and can use this in various ways.

WHAT IF I DON'T KNOW AHEAD OF TIME?

Sometimes you can be conservative and declare a constant length that really is the largest value permitted.

Another option is to pass MAXD in as a compiler argument!

```
g++ -std=c++20 –DMAXD=value myprog.cpp –o myprog
```

Many companies do this for things like photo dimensions where they have a general purpose program and want to specialize it for common photo sizes to get a better quality of code from C++

HOW DOES C++ LEVERAGE THIS CONST?

… for example, consider

```
myvec[MAXD-k-1] = c;
```

```
movb      %rbx,_myvec(999-%rax)
```

This sets the item “k” from the end to 8. **C++ can compute MAXN-1 as a constant**, and index directly to this item as an offset relative to myvec.

By having c and k in registers, only a single instruction is needed!

WHY IS THIS SO GREAT?

If C++ had not been able to anticipate that these are constants, it would have needed to *compute* the offset into digits.

- That would require more instructions.

Here, we are leveraging knowledge of (1) which items are constants, and also (2) that C++ puts "frequently accessed" variables in registers.

MORE EXAMPLES USING "CONST"

We can mark an argument to a method with "const".

This means "this argument will not be modified".

- C++ won't allow that argument to be used in any situation where it might be modified.
- C++ will also leverage this knowledge to generate better code.
- But the argument must correspond to a variable in memory

MORE EXAMPLES USING "CONST"

We can mark an argument to a method with "const".

```
// constant_values1.cpp
int main(const int argc, const char** argv) {
   const int i = strlen(argv[0]);
   ....
}
```

This means '

- C++ won' it
  might be n
- C++ will also leverage this knowledge to generate better code.

MORE EXAMPLES USING "CONST"

Illegal: For loop modifies n (C3892)

We can … an argument to a method with "const".

```
// constant_values2.cpp
int main(const int argc, const char**argv) {
   for(const auto n = argc; n < argc; n++)
        printf("%d'th argument is %s\n", n, argv[n]);
}
```

This means '

- C++ won' … it might be n
- C++ will also leverage this knowledge to generate better code.

WORTH KNOWING

const applies to everything on the right of it.

- const int& x: x is an alias for the argument, and it won't be modified. *Note that the argument passed in cannot be a constant in this case, like 25. An constant value (25) <u>doesn't live in memory </u>so we can't make a reference to it or take its address.*
- const int* x: x is a pointer to an int that won't be modified
- int* const x: x is a pointer that won't be modified, but it points to an integer that can be modified.

MORE EXAMPLES USING "CONST"

We can mark an argument to a method with "const".

This means '

- C++ won'
  might be r
- C++ will

```
// constant_values3.cpp
int main(const int argc, const char**argv) ) {
   char *buf0 = (char*)malloc(10);
   char *buf1 = (char*)malloc(20);
   const char* aptr = buf0;  // Initializes aptr...
   aptr[0] = 'a';            // OK
   aptr[1] = 'b';            // OK
   aptr = buf1;              // C3892
}
```

MO

We

This

- C
  m
- C

```
// constant_member_function.cpp
class Date
{
private:
   int month, day, year;

public:
   Date(const int&, const int&, const int&);
   Date(const Date&);                    // A "copy constructor"
   int getMonth() const;                 // A read-only function
   void setMonth(const int&);            // Updates month
};
```

ASIDE

The “const” suffix for a read-only method like getMonth *can only appear inside a method declared as a member of a class.* It means “read only property” of the object the class defines.

If you used this same notation on a global method, it will be rejected with an error message.

… BUT CONST CAN ALSO MEAN "I DON'T CHANGE THIS ARGUMENT"

In this sum function, we are saying "sum will treat **a** and **b** as constants (it won't change them). It accesses them by reference, so you cannot pass a constant to it

```
int sum(const int &a, const int &b) const
{   return a+b;
}
```

The const at the end is only permitted if sum is a method in some class. It says that this method will not change *member variables* in the class that defined it.

BY-REFERENCE SOMETIMES MATTERS A LOT!!

Consider this buggy code. litter is a field of type std::list<Kitten>

```
void addKitten(Cat c, Kitten k) {
        c.litter += k;
}
```

Where is the mistake?

BY-REFERENCE SOMETIMES MATTERS A LOT!!

… that version was modifying a *copy* of Cat c! To modify the original:

```
void addKitten(Cat& c, Kitten& k) {
        c.litter += k;
}
```

This version modifies the *original* Cat c. It also avoids making an extra copy of kitten k.

BY-REFERENCE SOMETIMES MATTERS A LOT!!

… even better is to explicitly say that Kitten k is constant:

```
void addKitten(Cat& c, const Kitten& k) {
        c.litter += k;
}
```

But notice that we can't say that Cat c is a const. We *update* c.

CONSTEVAL AND CONSTEXPR

The consteval keyword says that "this expression should be entirely constant". *The expression can even include function calls.*

C++ will complain if for some reason it can't compute the result at compile time: a constant expression turns into a "result" during the compilation stage.

If successful, it treats the result as a const.

CONSTEVAL AND CONSTEXPR

In contrast, the constexpr keyword says "try to evaluate this at compile time, but runtime code is ok if the evaluation fails."

C++ will not complain if your constexpr is not, in fact, a constant expression. It just creates some code and evaluates at runtime.

But, if successful, it treats the result as a const.

CONSTEVAL AND CONSTEXPR

The consteval keyword says that “this expression should be

e

```
constexpr float x = 42.0;
constexpr float z = exp(5, 3);
constexpr int i; // Not an error… but not a constant!  I isn’t initialized
int j = 0;
constexpr int c = j + 1; //Legal, but will be computed at runtime
consteval int k = j + 1; //Error! j isn’t const, so can’t be fully evaluated
```

C

a

th

If successful, it treats the result as a const.

FUNCTIONS USED IN CONSTANT EXPRESSIONS

To use a function in as an initializer for a const, or in a constexpr, the function itself must be marked as a constexpr.

The compiler will complain if any aspect of the function cannot be fully computed at compile time.

WE CAN COMBINE THESE ANNOTATIONS

Here we declare that exp is a constant expression using a recursive method to compute x^n

```
constexpr float exp(const float x, const int n)
{
   if(n == 0)
       return 1;
    if(n % 2 == 0)
       return exp(x * x, n / 2);
    return exp(x * x, (n - 1) / 2) * x;
}
```

HOW DOES THIS IMPACT FIBONACCI(N)?

If n is a constant, fibonacci(n) *can actually be computed as a constant expression too.*

The C++ consteval concept focuses on this sort of optimization. If something is marked as a consteval, C++ computes it at compile time, and gives an error if this fails!

Constexpr is very similar but with no error message if it fails. It just generates normal code if needed.

WE CAN COMBINE THESE ANNOTATIONS

C++ can compute fibonacci(5) as a constexpr entirely at compile time. It will just turn this into the constant 5. Same with consteval… but only if you supply a constant argument.

```
constexpr int fibonacci(const int n)
{
    return n <= 1? n: fibonacci(n-1)+fibonacci(n-2);
};
```

INLINE

```
inline int sum(const int &a, const int& b)
{
    return a+b;
};
```

Inline tells C++ to "expand" the code of the method. For example:

```
c = sum(a, b);
```

would expand into

```
c = a + b;
```

INLINING IS AUTOMATIC… YET THE KEYWORD IS STILL COMMONLY USED

In effect, when we write “inline” we often are giving a hint both to the compiler (which probably ignores the hint and makes its own decision!) and also to other readers of the code.

We are saying “*I wrote this code as a method, but in fact I am anticipating that this is really a “code pattern” that will be expanded for me, then optimized in place*”.

CONSTEVAL WILL SAVE 255 INSTRUCTIONS!

A big win for Fibonacci, as long as the compiler can actually compute the desired value at compile time.

If it can’t, the value isn’t a legitimate consteval. For the constexpr case, C++ will try inlining your code (and you can “force” it)

A CONCRETE PUZZLE

Consider this program:

Why doesn't inline cause an infinite recursion in the compiler?

```
#include <iostream>
using namespace std;

inline int fibonacci(const int n)
{
        return (n<=1)? n: fibonacci(n-1)+fibonacci(n-2);
};

int main(const int argc, const char** argv)
{
        for(int n = 1; n < 10; n++)
        {
                cout << "fibonacci(" << n << ") is " << fibonacci(n) << endl;
        }
        return 0;
}
```

INFINITE RECURSION???

In a language like C, which has a #define for inlining, the preprocessor expands #define before the compiler really runs

But this means the expansion doesn't have any way to notice that it is expanding Fibonacci(1) and Fibonacci(0).

… so if this was C #define, it would overflow and give an error

RECURSIVE INLINING?

In principle, if we call this version of fibonacci with a constant, it "should" expand it fully, then collapse the expression by realizing that constant arithmetic suffices.

But in fact when the compiler inlines a call to Fibonacci(0) or Fibonacci(1), it evaluates the if statement and doesn't do a recursive expansion!

RECURSIVE INLINING?

Now, suppose we call Fibonacci(10).

In principle, C++ can compute this and substitute the actual value (55) as a const!

In practice, the compiler limits how much recursion it is willing to do at compile time. It “gives up” and generates normal code if the constexpr task is too difficult.

OVERFLOWS?

Consider Fibonacci(100) = 354224848179261915075. This is too large for a 32-bit int, which is limited to 4294967295!

If C++ sees an integer overflow during consteval expansion, it gives an error code.

WHAT ABOUT THE FOR LOOP?

In a for loop from n=1 to 10, C++ can "tell" that the for loop iterates over 10 constant values: 1, 2, … 10

This enables it to "unroll" the loop and substitute a constant for each instance.

Will it do this? For fancy tasks, C++ can be unpredictable.

HOW DO THESE FEATURES “INTERPLAY” WITH VECTORIZATION?

To write code that will vectorize nicely, it is very important that the compiler can determine:

Sizes of your vectors and matrices

Loop “stride” values: The increment in a for loop

Expressions used to access matrix or vector elements

Values used to “map” from some input x to mapped[x]

For such purposes, constexpr arithmetic can be incredibly useful!

CONCEPT: STATIC ANALYSIS

Modern computing environments often include tools that do some form of analysis of programs or other objects before the execution actually occurs.

For the C++ compiler, constexpr and inline illustrate forms of static analysis that benefit the compilation stage.

HOW STATIC ANALYSIS IS DONE

Focusing on the C++ compiler, it first scans your program and forms a parsed code representation based on applying the syntax rules.

Next, it can study this graph structure to learn things.

What sorts of things can static analysis discover?

MORE STATIC ANALYSIS OPPORTUNITIES

We saw constants, arguments by reference and inlining

Static analysis might also discover loop bounds, “dead” code (an if statement that is never true, or always true), variables that do or do not need space allocated, etc.

Static analysis is also at the core of type checking.

CONSIDER THE “AUTO” DECLARATION

In C++ we often ask the compiler to figure out types:

```
std::map<std::string, Bignum> the_map;
…
for(auto item: the_map) {
        cout << “The next item is “ << item.to_string() << endl;
        do_something(item);
}
```

Here we created a map from string “names” to Bignum objects, then iterate through the map (item will be a sequence of std::pair objects)

CONSIDER THE “AUTO” DECLARATION

In C++ we often ask the compiler to figure out types:

auto requires a form of constexpr computation

```
std::map<std::string, Bignum> the_map;
...
fo  auto  m: the_map) {
        cout << “The next item is “ << item.to_string() << endl;
        do_something(item);
}
```

Here we created a map from string “names” to Bignum objects, then iterate through the map (item will be a sequence of std::pair objects)

EXAMPLE OF AN AUTO-DISCOVERED TYPE

When creating my “Bignum” solution, I once ran into this:

```
std::pair<typename std::_Rb_tree<_Key, std::pair<const _Key, _Tp>, std::_Select1st<std::pair<const _Key, _Tp> >,
_Compare, typename __gnu_cxx::__alloc_traits<_Allocator>::rebind<std::pair<const _Key, _Tp> >::other>::iterator,
bool> std::map<_Key, _Tp, _Compare, _Alloc>::insert(const value_type&) [with _Key =
std::__cxx11::basic_string<char>; _Tp = Bignum; _Compare = std::less<std::__cxx11::basic_string<char> >; _Alloc =
std::allocator<std::pair<const std::__cxx11::basic_string<char>, Bignum> >; typename std::_Rb_tree<_Key,
std::pair<const _Key, _Tp>, std::_Select1st<std::pair<const _Key, _Tp> >, _Compare, typename
__gnu_cxx::__alloc_traits<_Allocator>::rebind<std::pair<const _Key, _Tp> >::other>::iterator =
std::_Rb_tree_iterator<std::pair<const std::__cxx11::basic_string<char>, Bignum> >; std::map<_Key, _Tp, _Compare,
_Alloc>::value_type = std::pair<const std::__cxx11::basic_string<char>, Bignum>]
```

☹

WHAT IN THE WORLD WAS THAT???

The first thing to know is that C++ often generates its own variables. To avoid name conflicts, it puts underscore characters (_) at the front.

The second thing to know is that a std::map has a "comparison" function and an iterator, which (in my case) were defaults.

And so… this was the complete type for std::map<std::string,Bignum>.

IN FACT, C++ WOULDN'T BE USEFUL WITHOUT TYPE INFERENCE!

Const and constexpr are "natural fits" for C++ because the compiler is already doing so much automatic inference.

These annotations simply advise the compiler to do what it wanted to do in any case!

… just a glimpse of the true complexity of modern languages

WE SAW CONSTANT EXPRESSION MATH IN LECTURE 8, TOO!

C++ depends upon it to recognize parallelizable logic.

In fact, even code rearrangement can be understood as a form of constant expression evaluation: the code is like an expression, and all the variant forms of it are “equivalent” representations

This conceptual insight is key to modern compilation…

BUT BEWARE: NOT EVERY STATIC ANALYSIS PROBLEM CAN BE SOLVED!

We already saw this with constexpr and inlining: recursion can exceed the limitations of the compiler.

In fact, static analysis can even run into "unsolvable" problems!

- Type inference (auto) is potentially undecidable. Even the decidable versions have high complexity. Auto normally is successful.
- But experts can construct cases in which C++ may not be sure what the type of a variable is… and will give an error

SUMMARY FROM TODAY

C++ has advanced features that permit compile-time code analysis, compile-time type inference, and compile-time expression evaluation. This even includes recursive functions!

When we use const / consteval / constexpr, we "control" the compiler, which lets us ensure that the optimized code will use specialized instructions or achieve other kinds of efficiencies.

We code in an elegant, high-level way yet can control the compilation process down to ensuring that C++ will make the choices we want.